Таблица 11.1. Сортировка вставкой: время выполнения различных версий программы

Программа	Объем (строк на языке С)	Время, нс
Сортировка вставкой (1)	3	$11{,}9n^2$
Сортировка вставкой (2)	5	$3{,}8n^2$
Сортировка вставкой (3)	5	$3{,}2n^2$

Третьей программе требуется несколько миллисекунд для сортировки n = 1000 целых чисел, треть секунды на n = 10 000 целых, и почти час на сортировку миллиона чисел. Скоро мы встретимся с программой, сортирующей миллион чисел меньше, чем за секунду. Если входной массив уже почти отсортирован, сортировка вставкой работает гораздо быстрее, поскольку все элементы сдвигаются лишь на небольшое расстояние. Алгоритм в разделе 11.3 данной главы основан именно на этом свойстве.

11.2. Простая быстрая сортировка

Этот алгоритм был впервые описан К. А. Р. Хоаром в его классической статье «Быстрая сортировка» (C. A. R. Hoare, Quicksort. Computer Journal, 5, 1, April 1962, p. 10–15). В этом алгоритме используется подход «разделяй и властвуй», уже упоминавшийся в разделе 8.3: *чтобы отсортировать массив, мы разделяем его на два подмассива и сортируем каждый из них рекурсивно.* Например, для сортировки массива из восьми элементов

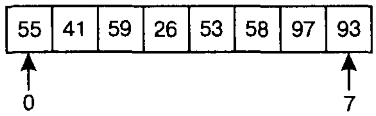

Рис. 11.1. Сортируемый массив

мы разбиваем его первым элементом (55), так чтобы все элементы, меньшие 55, расположились слева от него, а все большие — справа.

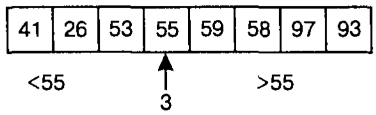

Рис. 11.2. Разбиение массива

Если затем рекурсивно отсортировать подмножества x[0..2] и x[4..7] по отдельности, весь массив окажется отсортирован.

Среднее время работы этого алгоритма оказывается существенно меньшим $O(n^2)$ сортировки вставкой, поскольку операция разбиения дает гораздо больше для достижения цели. После разбиения n элементов примерно половина из них окажется ниже выделенного элемента, а половина — выше. За то же самое время в программе сортировки вставкой на свое место устанавливается один-единственный элемент.

Итак, у нас есть набросок рекурсивной функции. Опишем рабочую часть массива с помощью индексов l, u (нижняя и верхняя границы). Рекурсия останавливается, когда мы добираемся до массива с числом элементов, меньшим двух. Код программы приведен на листинге 11.3.

Листинг 11.3. Быстрая сортировка (набросок)

```
void qsort(l, u)
  if l >= u then
    /* не более одного элемента - ничего не делаем */
    return
  /* цель: разбить массив относительно какого-либо элемента, который
оказывается на правильном месте */
  qsort(l, p-1)
  qsort(p+1, u)
```

Для разбиения массива относительно значения t мы начнем с простой схемы, о которой я узнал от Нико Ломуто. Более быстрый вариант программы, выполняющий эту задачу, будет приведен в следующем разделе[1], но эта функция так проста, что в ней трудно ошибиться, и она ни в коем случае не может быть названа медленной. Пусть дано значение t, нужно упорядочить массив x[a..b] и вычислить индекс m такой, что все элементы, меньшие t, находятся слева от элемента x[m], а все большие — справа. Мы решим эту задачу с помощью простого цикла for, сканирующего массив слева направо, используя переменные i и m для сохранения инварианта (рис. 11.3).

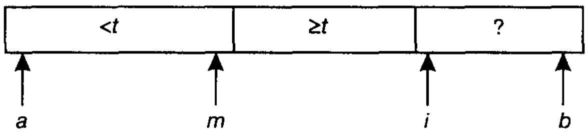

Рис. 11.3. Инвариант цикла сканирования в алгоритме быстрой сортировки

Когда проверяется i-й элемент, необходимо рассмотреть две ситуации. Если x[i] >= t, то инвариант остается истинным. Если же x[i] < t, то восстановить инвариант можно, увеличив индекс m (который указывает новое местоположение меньшего элемента) и поменяв местами x[i] и x[m]. В итоге получаем код, осуществляющий разбиение массива:

```
m = a-1
for i = [a, b]
  if x[i] < t
    swap(++m, i)
```

[1] В большинстве реализаций быстрой сортировки используется двустороннее разделение, описанное в следующем разделе. Хотя основная идея этого кода проста, детали реализации всегда казались мне весьма сложными — однажды я провел почти два дня в поисках ошибки в коротком цикле разбиения. Один из читателей черновика этой статьи пожаловался, что стандартный метод на самом деле проще, чем метод Ломуто, и в подтверждение этому набросал код программы. Я прекратил изучение этой программы, найдя две ошибки.

В алгоритме Quicksort нам нужно разбить массив x[l..u] относительно значения t=x[l], так что a = l + 1, a b = u. Инвариант цикла разбиения можно будет изобразить следующим образом (рис. 11.4).

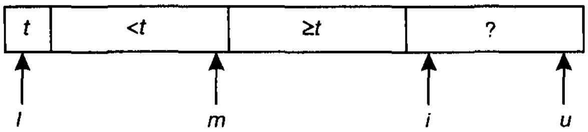

Рис. 11.4. Инвариант цикла разбиения массива

После завершения цикла получим картину, изображенную на рис. 11.5.

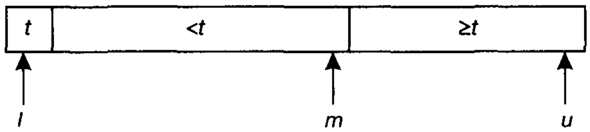

Рис. 11.5. Состояние массива после завершения цикла разбиения

Затем мы поменяем местами x[l] и x[m], что даст рис. 11.6[1].

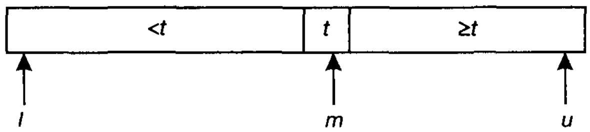

Рис. 11.6. Состояние после перестановки элементов

Теперь можно рекурсивно вызвать функцию с параметрами (l, m-1) и (m+1, u).

Получившийся код дает первую завершенную версию быстрой сортировки. Он приведен в листинге 11.4. Для сортировки массива x[n] функцию следует вызвать как qsort (0, n-1).

Листинг 11.4. Быстрая сортировка: версия 1

```
void qsort(l, u)
  if (l >= u)
    return
  m = l
  for i = [l+1, u]
    /* инвариант: x[l+1..m] < x[l] &&
    x[m+1..i-1] >= x[l] */
    if (x[i] < x[l])
      swap(++m, i)
  swap(l, m)
```

[1] Кажется соблазнительным пропустить этот этап и вызвать рекурсию с параметрами (l, m) и (m+1, u). К сожалению, при этом мы попадаем в бесконечный цикл, если t является наибольшим элементом массива. Возможно, я бы обнаружил эту ошибку в процессе проверки условия завершения, но читатель может догадаться, как я в действительности на нее наткнулся. М. Джейкоб дал элегантное доказательство некорректности такой программы: поскольку x[l] никогда не переставляется, эта сортировка будет работать только в том случае, если x[0] будет минимальным элементом массива.

```
/* x[l..m-1] < x[m] <= x[m+1..u] */
qsort1(l, m-1)
qsort1(m+1, u)
```

В задаче 2 в конце главы описана модификация алгоритма разбиения, предложенная Бобом Седжвиком (Bob Sedgewick). Она дает несколько более быстрый алгоритм qsort2.

Правильность этой программы была практически полностью доказана при ее выводе (как и должно быть). Доказательство ведется по индукции. Внешний оператор if правильно обрабатывает пустые и одноэлементные массивы, а алгоритм разбиения правильно подготавливает массивы большего размера для последующих рекурсивных вызовов. Программа не может войти в бесконечную последовательность рекурсивных вызовов, поскольку элемент x[m] при каждом таком вызове исключается. Таким же образом была доказана конечность алгоритма в разделе 4.3 (двоичный поиск).

Программа быстрой сортировки работает за время $O(n \log n)$ и требует в среднем $O(\log n)$ памяти на стеке. Под «средним» случаем подразумевается случайная перестановка не равных друг другу элементов, поступающая на вход алгоритма. Математические аргументы в пользу этого вывода аналогичны алгоритмам, приведенным в разделе 8.3. В большинстве учебников по теории алгоритмов подробно анализируется время выполнения быстрой сортировки. Кроме того, в них доказывается, что любая основанная на сравнении сортировка не может выполнить менее $O(n \log n)$ сравнений, поэтому быстрая сортировка наиболее близка к идеалу.

Функция qsort1 — это самая простая среди известных мне версия алгоритма быстрой сортировки. Она иллюстрирует важнейшие свойства этого алгоритма. Во-первых, он действительно быстр: в моей системе миллион случайных целых чисел упорядочивается чуть больше, чем за секунду, — вдвое быстрее, чем хорошо отлаженная библиотечная функция qsort языка С (последняя предназначена для сортировки различных типов данных, поэтому оказывается более медленной). Эта сортировка может быть удобной для некоторых приложений с хорошими входными данными, но она обладает и другим характерным свойством быстрых сортировок: для некоторых типов входных данных время ее работы может быть квадратичным. В следующем разделе рассмотрены более робастные быстрые сортировки.

11.3. Улучшенные быстрые сортировки

Функция qsort1 быстро сортирует массив случайных чисел, но что, если на вход будет подана уже упорядоченная последовательность? Как мы видели в разделе 2.4 главы 2, программисты часто используют сортировку для того, чтобы одинаковые элементы оказались рядом. Следовательно, нужно рассмотреть крайний случай: массив из n одинаковых элементов. Сортировка вставкой работает на таких данных очень быстро: каждый элемент сдвигается на 0 позиций, поэтому время выполнения растет как $O(n)$. Функция qsort1 справляется с такими данными очень плохо. Каждое из n-1 разбиений требует время $O(n)$ для выделения одного элемента, поэтому полное время выполнения растет как $O(n^2)$. Время обработки для n = 1 000 000 возрастает с одной секунды до двух часов.

Можно обойти эту проблему, используя двусторонний алгоритм разбиения с приведенным на рис. 11.7 инвариантом.

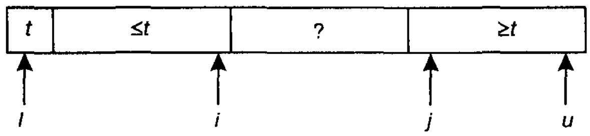

Рис. 11.7. Инвариант двустороннего разбиения

Индексы i и j инициализируются граничными индексами разбиваемого массива. Главный цикл содержит два вложенных цикла. Первый вложенный цикл сдвигает i вверх, пропуская меньшие элементы, а второй увеличивает j, пропуская большие элементы и останавливаясь на меньшем. Главный цикл проверяет, не пересекаются ли эти индексы, и переставляет соответствующие элементы.

Но как такой код будет работать в ситуации, когда все элементы равны? Первая мысль: пропустить эти элементы, чтобы не делать лишней работы, но в результате получается квадратичный для массива из одинаковых элементов алгоритм. Поэтому каждое сканирование будет останавливаться на одинаковых элементах, которые затем будут обмениваться. Хотя в этом варианте обменов будет производиться больше, чем требуется, такая программа будет превращать худший случай с массивом из одинаковых элементов в лучший, требующий почти в точности $n\log_2 n$ сравнений. Программа, реализующая описанный алгоритм разбиения, приведена в листинге 11.5.

Листинг 11.5. Быстрая сортировка (версия 3)

```
void qsort3(l, u)
  if l >= u
    return
  t = x[l]; i = l; j = u+1
  loop
    do i++ while i <= u && x[i] < t
    do j-- while x[j] > t
    if i > j
      break
    swap(i, j)
  swap(l, j)
  qsort3(l, j-1)
  qsort3(j+1, u)
```

Избавляясь от квадратичного поведения в худшем случае, этот код и в среднем делает меньше обменов, чем qsort1.

Рассмотренные нами программы быстрой сортировки разбивали массив относительно первого встреченного элемента. Это хорошо подходит для случайных входных данных, но может сильно замедлить работу для некоторых упорядоченных последовательностей. Если массив уже отсортирован по возрастанию, его придется разбивать относительно первого элемента, затем относительно второго и так далее, что потребует времени $O(n^2)$. Мы можем избежать этого, выбирая элемент для разбиения случайным образом — обменивая местами элемент x[l] со случайным элементом из диапазона x[l..u]:

```
swap(l, randint(l, u))
```

Если у вас нет функции randint, обратитесь к задаче 2 из главы 12 данной книги, которая посвящена написанию собственного генератора случайных чисел. Каким бы кодом вы ни пользовались, внимательно проследите за тем, чтобы функция randint возвращала значение из диапазона [l, u] — выход за его границы приведет к ошибкам. Объединив случайный выбор центрального элемента с двусторонним разбиением, мы получим программу быстрой сортировки, работающую за время $O(n \log n)$ для любого входного массива. Усреднение делается вызовом генератора случайных чисел, а не анализом возможного распределения входных данных.

Наша программа быстрой сортировки бо́льшую часть времени тратит на сортировку очень маленьких подмножеств. Такие массивы было бы проще всего сортировать каким-либо несложным методом вроде сортировки вставкой, а не тратить на них всю мощь быстрой сортировки. Боб Седжвик разработал весьма хитроумную реализацию этой идеи. Когда функция быстрой сортировки вызывается для небольшого массива (то есть l и u близки), она не делает ничего. Реализуется это путем замены первого оператора if нашей функции на следующий код:

```
if u-l> cutoff
  return
```

Здесь cutoff — некоторое небольшое целое число. После завершения работы функции массив будет не отсортирован до конца, но разбит на небольшие группы случайно упорядоченных элементов, причем все элементы одной группы будут меньше любого элемента всех групп, расположенных справа от данной. Сортировать элементы внутри групп нужно каким-то другим методом, и тут лучше всего подходит сортировка вставкой, поскольку массив уже почти упорядочен.

Для решения задачи сортировки целиком придется выполнить два вызова:

```
qsort4(0, n-1)
isort3()
```

Задача 3 посвящена выбору оптимальной величины порога cutoff.

На последнем этапе оптимизации программы можно раскрыть вызов функции swap во внутреннем цикле (поскольку другие два вызова swap лежат вне внутреннего цикла, их раскрытие не даст ощутимого результата). Последняя версия программы Quicksort приведена в листинге 11.6.

Листинг 11.6. Быстрая сортировка: версия 4 (итоговая)

```
void qsort4(l, u)
  if u - l < cutoff
    return
  swap(l, randint(l, u))
  t = x[l], i = l, j = u+1
  loop
```

```
    do i++ while i <= u && x[i] < t
    do j-- while x[j] > t
    if i > j
      break
    temp = x[i]; x[i] = x[j]; x[j] = temp
  swap(l, j)
  qsort4(l, j-1)
  qsort4(j+1, u)
```

Задачи 4 и 11 посвящены дальнейшему улучшению производительности быстрой сортировки.

В табл. 11.2 приведены сводные данные по всем версиям быстрой сортировки. Правая колонка указывает среднее время работы в наносекундах, требуемое для сортировки массива из n случайных целых чисел. Многие алгоритмы могут вести себя как квадратичные для некоторых конкретных входных данных.

Таблица 11.2. Быстрая сортировка

Программа	**Объем кода (строк языка С)**	**Время, нс**
Библиотечная функция языка С qsort	3	$137n \log_2 n$
Быстрая сортировка 1	9	$60n \log_2 n$
Быстрая сортировка 2	9	$56n \log_2 n$
Быстрая сортировка 3	14	$44n \log_2 n$
Быстрая сортировка 4	15+5	$36n \log_2 n$
Библиотечная функция С++ sort	1	$30n \log_2 n$

Функция qsort состоит из 15 строк быстрой сортировки и 5 строк сортировки вставкой. Для миллиона случайных чисел время выполнения лежит в диапазоне от 0,6 с для библиотечной функции sort языка С++ до 2,7 с для библиотечной функции qsort языка С. В главе 14 мы изучим алгоритм, гарантированно сортирующий n целых чисел за $O(n \log n)$ для любых входных данных (даже в худшем случае).

11.4. Принципы

Это упражнение дает нам несколько ценных уроков о сортировке в частности и программировании вообще.

Библиотечная функция qsort

Библиотечная функция qsort проста в использовании и работает достаточно быстро. Она работает медленнее, чем самодельные версии быстрой сортировки, только потому, что предназначена для применения к различным типам данных, поэтому сравнение элементов осуществляется через вызов внешней функции. Интерфейс функции sort языка С++ существенно проще: массив x сортируется вызовом sort(x, x+n). Кроме того, она достаточно эффективно реализована. Если системная